simple-tester, a testing tool for embedded Forth systems

Ulrich Hoffmann and Andrew Read

EuroForth 2019

Introduction

simple-tester is a very lighweight testing tool designed to assist the devlopment of a Forth system on embedded target. simple-tester's is inspired by the ANS Forth test harness [1]. One innovation is the use of hashing rather than memory to compare actual and expected results.

The need for a simple testing tool

There is a chicken and egg situation with any testing tool that is implemented within the system it is designed to test:

1. if there are bugs in the system then the testing tool itself may not function
2. if the testing tool is complex then it cannot be implemented until the system has largely been completed, so the testing tool is not avilable during the development phase
3. if bugs in the system happen disable charater I/O, then the testing tool will not be able to communicate test diagnosis

For these and other reasons we believe Test Driven Development [2] has not typically been applied to the development of Forth implementations on embedded systems, while the ANS Forth test harness is mainly used for verificaton at the final stage.

The goal of simple-tester is to:

1. allow unit testing on embedded targets with limited resources and
2. allow testing as early on as possible in the lifecycle of new forth systems, even before the system knows how to compile new colon word defintions

Test Driven Development of a Forth system

We present an illustration from seedForth [3] [4], which is an approach to developing an embedded Forth system without a cross-compiler. (Roughly speaking, a tokenizer running on the host compiles source code to a token file which is processed on the target.) In the example below we bring up a series of elementary code words in seedForth and test them.

```
Tstart
  CODE: drop
  T{ 1 2 drop }T 1 ==

  CODE: dup
  T{ 1 dup }T 1 1 ==

  CODE: swap
  T{ 2 1 swap }T 1 2 ==
Tend
```

`CODE:` is part of seedForth, not simple-tester. (Briefly put, it simply "activates" a code word that has been implemented in assembly language on the target.)

`Tstart` is part of simple-tester and we assume that it, like the rest of simple-tester, is already implemented as a code word on the target. `Tstart` initiates testing.

`T{` designates the start of a test. The syntax is the same as the ANS Forth test harness.

`}T` designates the end of the section of code being tested. After `}T` there follows the expected results. This is different syntax to the ANS test harness.

`==` compares the actual results with the expects results and takes action if they do not match. This is new syntax from the ANS test harness, but arguably the post-fix comparison is more Forth-like. (And also avoids the `->` operator that is used for assignements in VFX.)

`Tend` concludes the series of tests.

simple-tester's communication protocol

simple-tester communicates entirely through a single numeric output device. This could be a line of micro LEDs (presenting numbers in binary), a seven-segment hexadecimal display, bytes over a serial line, or some other mechanism.

At the inception of testing `Tstart` sets the internal test counter to zero. At the start of each test, `T{` increments the internal test counter and reports the test number on the numberic output device. At the conclusion of each test, `==` acts as follows: if the actual results match the expected results, then do nothing. Otherwise halt the system whilst leaving the current test number visible on the numeric output device.

Assuming that all tests conclude sucessfully, then `Tend` displays some magic number, typically `FFFF`, to indicate successful conclusion. On the other hand if any test has failed then `Tend` will not be reached and the sequential number of the failing test will remain on the numeric output device.

This testing protocol was chosen for the following reasons:

1. we leverage hardware on the embedded system for output rather than rely on high-level Forth words such as "dot". Outputting a number to a line of LED's or a seven-segment display can often be accomplished with a single store instruction
2. implementing this communication protocol is very simple and requires minimal code
3. the test number is displayed before the code under test is executed. If the execution of the code causes a system-failure, then the identity of that test will already have been reported
4. assuming that all tests have completed sucessfully, then `FFFF` on the numeric output device is quickly and conveniently noted

This communication protocol is more limited than that of the ANS test harness: the reason for a test failure (different stack count or different actual results) is not reported, and only the first failing test is identified, not the full set. Nevertheless we consider that the advantages listed above are compelling in the situations where we envisage using simple-tester.

Implementing simple-tester

We highlight the key aspects of implementation before walking through the reference implementation in the next setion.

simple-tester is implemented as code words (likely in target system assembly language), rather than as colon definitons. Ideally these code-words should be implemented at a very early stage so that they can be employed to test further code words as they are developed.

We note that the ANS test harness stores the actual results of each test in memory prior to comparison with the expected results. We consider this approach to be less suitable for embedded systems since RAM may be limited. Instead we use a simple hash algorithm to hash both the actual and expected results and compare only the hash totals. Using this approch, our reference implementation requires only two cells of RAM storage.

We recognize that a hash approach may lead to false test passes where the actual and expected results are different but where there is a collision between the hash totals. We don't consider this weakness to be fatal - few testing approaches provide complete coverage of all possible cases, and so a judgement must always be made as to what level of testing coverage is sufficent to provide the required level of assurance.

We do take the precation of using a hash algorithm that is non-symmetric: if the actual and expected test results are the same values but in reversed order, then the test will fail. Of course a more sophisticated hash algorthim than the one we have chosen may be implemented if hash collisions are anticipated to be a problem in any particular situation.

Reference implementation

Although simple-tester is anticipated to be implemented as code words rather than in colon definitions, the reference implementation is given in Forth for ease of communication.

```
\ utility words
\ report the test number to a numeric output device
: T.
    .       \ for gforth testing
;

\ halt the system
: halt
    quit    \ for gforth testing
;

\ compute h1 by hashing x1 and h0
: hash ( x1 h0 -- h1 )
    swap 1+ xor
;

\ hash n items from the stack and return the hash code
: hash-n ( x1 x2 ... xn n -- h )
    0 >R
    BEGIN
        dup 0 >
    WHILE
        swap R> hash >R
        1-
    REPEAT
    drop R>
;

variable Tcount     \ the current test number
variable Tdepth     \ saved stack depth

\ start testing
: Tstart
    0 Tcount !
;
```

```
\ start a unit test
: T{ ( -- )
    Tcount @ 1+ dup T. Tcount !
    depth Tdepth !
;

\ finish a unit test,
: }T ( y1 y2 ... yn -- hy )
    depth Tdepth @ -    ( y1 y2 ... yn Ny )
    hash-n              ( hy )
    depth Tdepth !      ( hy )
;

\ compare actual output with expected output
: == ( hy x1 x2 ... xn -- )
    depth Tdepth @ -    ( hy x1 x2 .. xn Nx )
    hash-n              ( hy hx )
    = 0= IF halt THEN
;

\ signal end of testing
: Tend  ( -- )
    65535 ( 0xFFFF) T.
;
```

Potential applications

simple-tester has been designed first and foremost with the goal of supporting Test Driven Development of Forth systems on embedded targets. simple-tester might also be useful in other situations, for examlple:

1. Test Driven Development of applications on embedded Forth systems. Our experience is that the ANS test harness is not commonly used on embedded systems because of its size and complexity. Where simple-tester is already built in to a new embedded Forth system, then there is no reason why it cannot also be employed application testing

2. power on self testing (POST). For example, in seedForth the Forth system is recompiled from a token file at each power on, and since unit tests are interleaved with the Forth system definition, the system is freshly tested at every restart. This helps ensure that that the system can never be modified without subsequent regression testing, and also verify any hardware upon which the system is reliant

3. as an alternative to the ANS test harness for small application development. Since the reference implemention of simple-tester is provided in Forth it could also be used for application testing on desktops. There may be occasions where the simpler and lighter simple-tester, although more limited in scope than the ANS test harness, might be easier to handle

Conclusion

We have developed a very lighweight tool for supporting Test Driven Development, with a particluar focus on Forth systems in embedded targets. simple-tester is open source and available on GitHub [5]. We welcome correspondence.

Ulrich Hoffmann (FH Wedel University of Applied Sciences), uh@fh-wedel.de

Andrew Read, andrew81244@outlook.com

References

[1] http://www.forth200x.org/documents/html/testsuite.html
[2] https://en.wikipedia.org/wiki/Test-driven_development [3]
http://www.complang.tuwien.ac.at/anton/euroforth/ef18/papers/hoffmann.pdf
[4] https://github.com/uho/preForth
[5] https://github.com/Anding/simple-tester